The ShaPE of ShaDe: a coordination system

S. Castellani       P. Ciancarini       D. Rossi

Technical Report UBLCS-96-5
March 1996

Department of Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Requests and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

94-20 Dynamic Allocation of Signature Files in Multiple-Disk Systems, P. Ciaccia, August 1994.
94-21 Parallel Independent Grid Files Based on a Dynamic Declustering Method Using Multiple Error Correcting Codes, P. Ciaccia, November 1994.
95-13 Optimal Multi-Block Read Schedules for Partitioned Signature Files, P. Ciaccia, August 1995.
The ShaPE of ShaDe: a coordination system

S. Castellani       P. Ciancarini       D. Rossi

Technical Report UBLCS-96-5

March 1996

Abstract

ShaDe is an object-based coordination language. It offers a basic abstraction called the Object Space, that is similar to a tuple space with the difference that it contains objects and messages. ShaDe objects are active, i.e., they are units (places) of computation. Each object encapsulates a state in form of multiset of tuples and methods in form of rewriting rules. The object space is a coordination medium supporting a number of inter-object associative communication mechanisms, namely unicast, multicast, and broadcast.

The most interesting feature of Shade is that coordination is expressed by rules. We exploit such a feature to build “coordination” services enacting declarative cooperation laws. We demonstrate the use of the language building two coordination applications, namely a distributed auction system and a stock exchange system.
1 Introduction

Several systems consisting of applications distributed over either local or geographical networks are already available today: for instance, e-mail systems, audio and video teleconferences, shared whiteboards, just to give some examples. This kind of software is needed for supporting communication and cooperation requirements of modern virtual enterprises and workflow-based production processes [19].

There is the need of programming models and languages powerful enough to support the design and the implementation of this kind of systems, that are typically distributed across several networked workstations. These models and languages should be able to express the basic abstract features of a distributed system: parallel execution, concurrent access to shared documents and resources, synchronous and asynchronous communication, and lack of global/centralized data and control structures. However, the most important features are open-endness and multi-user cooperation and coordination [1, 29].

Coordination models and languages based on shared dataspaces [8] are gaining momentum as languages for designing and programming open multiuser software systems [11]. For instance, Interaction Abstract Machines (IAM) [5] is a coordination model which allows to express these fundamental features for structuring a distributed system. IAM is based on multiset rewriting; the model includes multiple multisets, also called multiple dataspaces. The first language to be based on IAM was LO [7].

Our goal in this paper is to define an object oriented coordination language for programming distributed systems, extending the IAM model. The language we have developed is called ShaDe, which stands for Shared Dataspaces. The dataspaces are objects, i.e. they encapsulate state and behavior, and there are operations to create new objects starting from existing objects (eg. inheritance, delegation, etc.). The objects themselves are shared in several ways: they are shared by agent threads inside them, they are shared by other objects, they are shared by users accessing the same object space.

In the current version the computation sublanguage used inside an object is Prolog, so that we say that ShaDE is a coordination language with object oriented features and a logic programming foundation. Such a combination is motivated by the fact that both the object oriented and the logic programming paradigms have several complementary features [12]. Logic programming features a sound and clean semantics. Moreover, its theoretical roots provide expressive power and implicit parallelism. On the other side, the object oriented paradigm allows to clearly define concepts like state change, modularity, and inheritance. Objects allow to structure programs in a clean and understandable way, reflecting the structure of the problem to be solved.

We are interested in all the aspects of distribution and parallelism that such a combination can offer. We have chosen then a “translation” approach, that is, we have specified in an object-oriented fashion a language inspired by the IAM model, which we have mapped in a logic language, namely Linear Objects (LO), which has a strong formal foundation in Linear Logic [16]. ShaDe itself is the evolution of COOLL [9].

In Sec.2 we motivate our work. In Sec.3 we introduce the ShaDe coordination language, describing its syntax and semantics. In Sec.4 we illustrate its current implementation, including a distributed run-time system and a programming environment called ShaPE. In Sec.5 we describe the design and implementation of two complex applications where coordination is the main problem, namely a distributed Auction System, and a Stock Market System, presenting their implementations in ShaDe. In Sec.6 we formalize ShaDe semantics and show how it is possible to map ShaDe programs onto LO programs, thus giving to ShaDe a semantics that is similar to the one offered by LO [6].

2 Motivation

The ubiquity of the Internet suggests its use for supporting applications involving remote cooperation and resource sharing. Such applications coordinate a dynamic number of users, which enter and leave these systems to cooperate and share services. The WWW is clearly showing
how effective can be such applications, even if its current scope simply consists of large scale information retrieval and exchange operations [13].

The goal of our work is to facilitate the construction and integration of distributed applications supporting coordination, cooperation, and communication, both synchronous and asynchronous.

In the following table we classify a number of applications according to the kind of basic mechanisms they need.

<table>
<thead>
<tr>
<th>communication</th>
<th>synchronous</th>
<th>asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IRC, teleconferencing</td>
<td>e-mail</td>
</tr>
<tr>
<td>cooperation</td>
<td>groupware (eg. co-authoring systems)</td>
<td>DBMS (ftp, WWW)</td>
</tr>
<tr>
<td>coordination</td>
<td>distributed game playing (MUD) auction system</td>
<td>workflow (eg. meeting scheduler)</td>
</tr>
</tbody>
</table>

Communication mechanisms allow to exchange messages and/or data streams; these are the basic services needed to build applications like IRC (Internet Relay Chat), e-mail, or teleconferencing systems [23].

Cooperation mechanisms allow to share documents and resources; groupware applications like coauthoring systems [14, 17] and shared data repositories need specific cooperation protocols, respectively synchronous or asynchronous.

Last but not least, coordination mechanisms allow the orchestration of multiple activities and services, useful to build applications like either distributed game-playing environments or workflow support systems. In this paper we develop a coordination language and show how it can be useful for designing coordination systems, both synchronous and asynchronous.

A basic choice we made was to build a language useful for designing and controlling systems made of distributed objects [25]. Software designers use object-orientation for convenience; they use parallelism for performance. When parallelism is combined with object-orientation, a software system is conveniently decomposed in a collection of active objects [2]. Active objects are concurrent agents which need coordination [1]. The interactions among active objects are often represented by message passing based on objects’ names or message channels as in [18], where CSP-based primitives are used. However, this is a low level coordination mechanism which is difficult to control in an open, distributed framework where agents can indifferently be clients or servers [19].

In fact, designers of open systems, which typically include programs and data to be distributed across many processors and repositories, have to solve several problems concerning issues like locating and disciplining access to resources, establishing communication links among running programs and coordinating their execution, synchronizing replicated data or programs to maintain a consistent state or to ensure fault-tolerance [24, 19].

The first generation of parallel/distributed OO languages and systems [10], such as Actors [3], POOL [4], and ABCL [31], offer either low level or too complex coordination mechanisms; moreover, users have often to learn a brand new programming language for computation.

The basic idea that “coordination is orthogonal to computation” [8] was the inspiration for introducing Shade. One of our main goals designing Shade is to create an open coordination space for objects. We wanted our system to be open to the outer world in order to ease the interaction with it. This way Shade can be used to coordinate not just software objects but also users interactions.

3 The coordination language ShaDe

An object is a self-contained unit, encapsulating both data and behaviour [28]. Encapsulation means that data contained in the object is visible only within the object itself. The behaviour of an object is described by its class, which defines the operations (methods) which can be invoked by sending a message to the object. Inheritance allows a class to be defined as a derivate of another class.
In ShaDe all these concepts are present, but with a special flavor. In fact, ShaDe objects are active, i.e. they are units (places) of computation, having both an internal -parallel- activity and the ability to react to the reception of messages sent by other objects. The state of an object is a multiset of tuples, while the object itself can be considered as a dataspace. Moreover, ShaDe objects can use several types of communication mechanisms: namely unicast, multicast, and broadcast.

In the following sections we illustrate ShaDe starting from its syntax.

3.1 Syntax
A ShaDe program is a set of classes. The behaviour of an object is defined by the methods of the class it belongs to. Classes are modules with a unique name in the program, and inheritance is obtained by methods union.

The definition of a class in ShaDe has the following form:

:class C\_name = \{method\_1 \# ... \# method\_n \}.

There is no declaration of state. In fact, the definition of a class only includes the definition of its name C\_name, and a set of methods. A method has the following form:

\textit{method} =
\begin{itemize}
  \item \textit{read} tokens;
  \item \textit{in} tokens;
  \item with \{ seq\_program \};
  \item \textit{send} messages;
  \item \textit{create} objects;
  \item \textit{out} actions
\end{itemize}

tokens = token, tokens | tokens
messages = message, messages | message
message = [dest, tokens]
dest = string
objects = object, objects | object
object = [object\_type, object\_name, tokens]
actions = tokens | terminate

where:
- \textit{tokens}: is a multiset of tuples;
- \textit{seq\_program}: is the call of a sequential program;
- \textit{messages}: is a sequence of communication operations separated by commas;
- \textit{dest}: is an identifier defined by a string;
- \textit{objects}: is a sequence of object definitions separated by commas;
- \textit{object}: is the definition of an object;
- \textit{terminate}: is the symbol of “termination”.

There are some constraints on the syntax:
- the \textit{in} entry cannot be empty;
- the \textit{in} and \textit{out} entry cannot contain the type of the object or its name.

The atom self represents the name of an object.

3.2 Identifiers, Names, and Types
A simple identifier is a string of letters and digits. A composite identifier is an expression of the form $p(t_1, \ldots, t_n)$, where $p$ is a simple identifier, $n > 0$, and $t_i$ can be a simple or a composite identifier.
The name of a class $C$ is the identifier that appears in the header of its definition. Such an identifier can be simple, like fruit for example, or a sequence of simple identifiers separated by ":", like fruit:apple.

The name of a prototype $P$ is the name of the class $C$, $C\_name$, of which the prototype is the image.

The type of an object $O$, instance of the prototype $P$, is the name of the prototype, $P\_name$.

The name of an object $O$, instance of a prototype $P$, whose name is $P\_name$, has the following form:

$P\_name(composite\_I)$

The names of the objects, instances of the same prototype, are distinguishable then by the part $composite\_I$.

The name of a class $C$, and of the corresponding prototype, can be a succession of simple identifiers, separated by ":". In this way we declare the class $C$ as a subclass of another class $C'$. The class $C'$ is then a superclass of the class $C$.

For instance, if we have the class fruit:apple, then this is a subclass of the class fruit.

This relationship between classes introduces a form of inheritance. ShaDe offers an "extending semantics" form of inheritance. The methods of a class are the result of the union among the methods of the superclasses, if any, and the methods directly defined for the class. This static form of inheritance can be resolved at compilation time.

### 3.3 Running a program

A programmer should imagine the operating environment where his ShaDe program runs as an object space made of "places" where several activities happen. Three types of entities are relevant: classes, prototypes, and objects.

Syntactically, a program is composed of a set of classes. When a program is run, one prototype for each class in the program is created. A prototype is the run-time image of a class, that is, a prototype is an active entity in charge of the creation of objects of that class. An object is an instance of a prototype (this is a form of delegation: see [22]), that is, each time a prototype receives a request for the creation of an object, the prototype creates an object whose behaviour is described by the methods of the prototype.

A prototype resides in the system until the program terminates, whereas objects can terminate during the program execution.

An object can be dynamically created either implicitly by another object or explicitly from outside (see Sec.3.5).

### 3.4 State Transitions of Prototypes and Objects

An object $O$ is an instance of a prototype. It is denoted by a name $N$, and a type $T$ (see Sec.3.2). An object $O$ has a "state" $S$ and a set of methods $R$.

The initial state $S_0$ of the object is configured when it is created, while its methods $R$ are the methods belonging to the class $T$. States are represented as multisets of tokens, whereas methods have the semantics of state transitions described by multiset rewritings.

As we said in Sec.3.1 a method has the following structure:

```plaintext
method =
    read tokens;
    in tokens;
    with \{ seq\_program \};
    send messages;
    create objects;
    out actions
```

A method describes a transition on the object state: a method is selected if the current state configuration satisfies a set of conditions. Once selected it commits and causes a change of the
state. The fields **read**, **in**, and **with** define the conditions that have to be satisfied for the selection of the method. The fields **send**, **create**, and **out** define the actions deriving from such selection.

More precisely, a method is selected if:

- **tokens** in **read** are a subset of the state;
- **tokens** in **in** are a subset of the state;
- **seq_program** in **with** terminates successfully;

Once the method has been selected the following events take place:

- **tokens** in **in** are removed from the state;
- **messages** in **send** are sent to the objects denoted as receivers;
- a request of creation for each object in **create** is sent to the system;
- if **actions** in **out** include **terminate**, the whole object terminates, else the tokens in **actions** are inserted in the state.

### 3.5 Coordination mechanisms

Communication in ShaDe is specified using the **send** field in a method. That is, each communication **message** has the form:

\[(dest, tokens)\]

The **dest** field is the **associative address** of the receiver whereas **tokens** are the contents of the communication.

An **associative address** is a string, which results in a regular expression. Objects recognize themselves as receivers if an instantiation of the string contained in **dest** corresponds to their name. In this way every object who recognizes itself as a receiver can access the message. The different ways of specifying such a field correspond to the following different modalities of communication:

- **broadcasting**. If we specify in the **dest** entry “_”, all the objects will receive such a message, because they all recognize themselves as receivers.

- **multicasting**. If we specify in the **dest** entry the name of a prototype, for example “fruit”, all the objects that are instances of the prototype whose name is fruit will receive this message, because they can recognize themselves as receivers, being of the type fruit.

- **unicasting**, i.e. **direct communication**. If we specify in the **dest** entry the name of an object, for example “fruit(1)”, all the objects that are instances of the prototype named fruit, and whose name is fruit(1), will receive this message because they can recognize themselves as receivers, having the name fruit(1).

The basic mechanism for all forms of communication is pattern-matching, that is, an object will receive a message if the name of the object matches the **dest** entry in the message.

New objects can be created at run-time in a ShaDe system. A creation request of a new object can be formulated to the system by an explicit request from outside or from another object that during its life, triggers a method with a non-empty **create** field. In both cases the request for the creation of an object is translated in a message directed to the ShaDe system of the following form:

\[create(objType, objName, tokens)\]

When a ShaDe system receives such a message, it delegates the interpretation of the message to the prototype **objType**. As a result the prototype creates an object with **objType** as type, with **objName** as name, and an initial state given by the contents of **tokens**.

A prototype terminates when the program terminates.

An object terminates if a method with the field **out** containing **terminate** is selected.

A program terminates when a termination request is explicitly made from outside.
4 A Distributed Implementation of ShaDe

4.1 Run-time

The ShaDe run-time system is composed of the Ether and several animators. The Ether is a coordination medium. The task performed by animators consists of animating objects. Each animator uses the Ether as described below. The main component of Ether is an object space where objects are stored once created, waiting to be animated by animators. The Ether also contains a reference space for addressing static objects linked to ShaDe.

An animator is essentially an object server which repeats forever the following loop: it grabs (consumes) an object from the object space, it applies a method (if this is possible), thus modifying the object state and/or communicating with other objects, then puts the object back in the space, and so on.

The Ether contains also the message space. When a message is generated by an object, it is put back inside the message space. The message space is accessed associatively by each object when no program method is applicable. Messages are hold until the program terminates; they are read by objects in the same order they are generated, starting from the first one, no matter when the object has been created.

Such a communication model is associative and persistent. Message reading on behalf of an object is performed by the Ether when an animator, which puts back an object in the objects space, notifies the Ether that no method was applicable. If no message is available for such an object it is put in the waiting objects space. When a new message is generated, each object which can read it is removed from the waiting objects space and put back in the object space.

The ShaDe system is open to the outer world, so access to the message space has been granted also to static objects. Each static object can send and receive messages. Messages sent by a static object are managed by the Ether just like any other message. Each time a message whose address matches some static objects’ address is generated, the message itself is delivered immediately to the matching static objects. This way we can integrate in the Ether external, independent agents, such as user interfaces and servers. Due to their particular nature the message management for static objects is quite different: in this case messages are associative but not persistent. We adopted this policy because static object are intended to be “hooked” to the Ether “on the fly” each time a user interaction or a service is needed.
Figure 2. The interface to write a Shade program

The Ether thus includes five main services: the messages space, the objects space, the waiting objects space, the prototypes objects space, and the static objects’ reference space, as shown in Fig.1.

The prototypes objects, due to performance considerations, are managed inside the Ether itself. There are, in fact, no real prototype objects and objects’ creation is managed using an ad hoc Ether call that supersedes the create message method formerly described.

4.2 The ShaDe Programming Environment

A basic ShaDe Programming development Environment, namely ShaPE, has been created so that a user can search, open, edit, and run a program. ShaPE is realized with tcl, tk, and expect [20, 21]. It currently includes the following modules:

env.etk: This is the main module. When it is invoked a window called “ShaDe” is displayed, and the four links new, edit, launch, exit are active. Selecting one of the links new, edit, or launch will invoke respectively newProgram.etk, browse.etk, launch.etk. Selecting the last link the user will exit ShaPE.

newProgram.etk: the invocation of this module displays a window that allows the user to enter a new program; the user can insert classes and methods of a new program (fig.2).

browse.etk: the invocation of this module displays the window “Browser”. Such a window visualizes a list of the files containing ShaDe programs. These can be searched selecting the menu “File” and then the menubutton “Search”. Programs can be removed selecting the menu “File” and then the menubutton “Remove”. When the user doubleclicks on the name
of a file, the window “Editor” is displayed. It contains a view that visualizes the contents of the selected file.

The user can search for a class in the program, selecting the menu “File”, and then the menu button “Search”. A class or a method can be selected by clicking on. In the first case the name of the class will be retrieved, and the fields of its first method will be extracted and visualized in a template. In the second case the fields of the selected method are extracted and visualized in the template.

The user can insert a new class or a new method in a selected class, selecting the button “AddClass”, and “AddMethod”, respectively.

Once selected, clicking on it, a class or a method can be removed selecting the button “Remove Class” or “RemoveMethod” respectively.

A method can be selected clicking on it, modified, and then updated selecting the button “Update”.

launch.etk: the invocation of this module displays the window “Launch”. This window provides the tools to launch a program (fig3).

A user can select one of the ShaDe programs contained in a given directory. Once a program has been selected, a ShaDe system can be created, that is, a program can be loaded. This
means that a prototype for each class of the program is created, and the communication server is initiated.
The user can then ask for the creation of a new object, and can send a message to a particular object.
The user can observe the messages sent during a computation in a view, selecting the mode “observer”, or can observe all the messages sent by the objects, and the error messages sent by the system selecting the mode “debugger”.

dialog.etk : this module is invoked when an error arises, it displays a message which communicates to the user the events that caused the error.

5 Some Sample Applications

Nowadays tools for supporting cooperation applications across networks are more and more requested. These applications can be as simple and widespread as e-mail systems, or as complex as groupware supporting the coordination of a given number of users and applications with a common goal (Active mail [17], Software Secretaries [26], etc.).

We have developed two applications to test our coordination system: a distributed Auction System, and a Stock Exchange. They are both applications where the coordination of several users and programs is the main issue.

5.1 Auction System

It is required to create a distributed Auction System accessible on network; the system should provide services for the following users [30, 15]:

- **Auctioneer**, who sells items to the best offer;
- **Participants**, who buy items by participating to the auction;
- **Observers**, who are (passive) auction spectators.

A network allows the auctioneer and the participants to communicate, during the bidding, and the observers to follow the bidding. We distinguish the following phases:

Phase 0 : The auctioneer makes the auction public via news. The network provides for an information package with the modalities for the users interested in the auction. Users can simply observe the bidding, or actively participate. In the first case, they will be informed of the bids, but they could not buy an item. In the second case, they could actively participate to the bidding, sending bids for an item, if at the beginning they have produced some credentials (a credit card number, for example).

Phase 1 : The auctioneer puts an item up for auction. A user connected to the auction is considered as an observer, and he becomes a participant sending the communication SUBSCRIBE. Users, who are interested in participating actively, send a communication SUBSCRIBE with the credentials. The SUBSCRIBE communication can be sent also during the bidding, it is then always possible to access the auction, and the participation is free. The auctioneer establishes the interval of waiting time for the reception of the participant communications, TIMEOUT, as long as he likes. Moreover, the auctioneer can send notification messages to all the users that are connected to the auction.

Once started the auction, the auctioneer collects the credentials of the initial participants during the time interval WAITSUBSCRIPTION.

Phase 2 : The auctioneer sends a communication to all the registered participants and the observers who are connected. The communication contains the CURRENTBID, which at this time is the BASEPRICE of the item.

Phase 3 : When the participant receives a communication containing the CURRENTBID for a given item, he can send one of the following answers:

- **NEWBID**: if the participant wants to send a new bid higher than the CURRENTBID;
- **UNSUBSCRIBE**: if the participant wants to give up.

Phase 4 : The auctioneer receives the answers of the participants until TIMEOUT. The auctioneer will react to the reception of such communications according to the following schema:
if the communication is NEWBID then the auctioneer compares NEWBID with CURRENTBID. If NEWBID is greater than CURRENTBID then NEWBID becomes the new value CURRENTBID, and the auctioneer sends a communication containing the new value for the CURRENTBID to all the participants and observers. The auctioneer then resets the time and the phase 4 restarts.

if the communication is UNSUBSCRIBE, the auctioneer eliminates the participant from the list of participants.

After TIMEOUT the auctioneer informs all the participants that the item was sold, if the value of CURRENTBID is higher than the value of the BASEPRICE. The auctioneer proceeds then putting on sale other items, if any (phase 1), or closing the auction. If the value of CURRENTBID is equal to the value of BASEPRICE the auctioneer archives the item as UNSOLD, and proceeds with the selling of other items, if any, otherwise he closes the auction.

All participants and observers can see the progress of the bidding. Moreover, the users have a choice:

- interactive participation on-line;
- automatic participation, using a “knowbot” agent able to represent the user in the auction using some predefined politics.

In the first case the user participates interactively to the bidding, receiving the current bids, and sending new bids. In the latter case the participant delegates a personal software agent (knowbot) as his representative; the system creates and supports the knowbot, while other participants have the illusion of competing with a normal (interactive) participant.

5.2 Auction System in ShaDe

In order to model the Auction System in ShaDe we introduce two classes, namely one for the Auctioneer and one for the Participants.

We present two solutions, which differ for a number of coordination features.

5.2.1 Solution with knowbots only, fixed politics

We present first a non-interactive solution, in which the system has only to coordinate the offers of knowbots.

The main Data Structures for the Auctioneer are:

- `waisubscription(Interval)`: this is the structure where the auctioneer stores the interval time for collecting the subscriptions.
- `credL(List)`: this is the structure where the auctioneer stores the credentials of the participants;
- `accPart(Part)`: this is a data that denotes a participant;
- `new_item(Name,MinB,MaxBid)`, `item(Name,MaxB)`: these structures contain the information about the item;
- `curr_bid(Name,Id,Bid)`: this structure represents the current bid for the item.

class auctioneer = {
  in time(Interval);
  out waisubscription(Interval)
  #
  in waisubscription(Interval), up_item(Name);
  send [_, timeForSubscription (Name, Interval)];
  out wait_cred (Name), credL([])
  #
  read wait_cred (Name);
  in subscribe (Part, Name, CreditNum), credL(L);
  send [_, ok (participant (Part), Name)];
  out credL([part (Part, CreditNum)|L]), accPart (Part)
in timeout(Name),wait_cred(Name);
out ok(Name)
#
in subscribe(Part,Name,CreditNum),credL(L);
send[_,ok(participant(Part),Name)];
out credL([part(Part,CreditNum)|L]),accPart(Part)
#
in unsubscribe(Part,Name),accPart(Part)
#
in new_item(Name,MinB,MaxB),ok(Name);
send[_,sell_item(Name)],[_,curr_bid(Name,nobody,MinB)];
out item(Name,MaxB),curr_bid(Name,nobody,MinB)
#
read item(Name,MaxB),accPart(WhoNew);
in new_bid(Name,WhoNew,NewB),curr_bid(Name,Who,CurrB);
with {NewB > CurrB};
send[_,curr_bid(Name,WhoNew,NewB)];
out curr_bid(Name,WhoNew,NewB)
#
read accPart(WhoNew);
in item(Name,MaxB),curr_bid(Name,Who,CurrB);
with {CurrB >= MaxB};
send[_,item_sold(Name,Who,CurrB)];
out item_sold(Name,Who,CurrB)
#
in item(Name,MaxB),curr_bid(Name,Who,CurrB),end(Name);
send[_,item_sold(Name,Who,CurrB)];
out item_sold(Name,Who,CurrB)
).

Class Auctioneer

Method1: The auctioneer sets up the time interval for the subscriptions of the participants.

Method2: The auctioneer communicates to the participants that they have the time interval Time to subscribe to the auction.

Method3: The auctioneer collects the subscriptions and stores the credentials of the participants.

Method4: When the time of the subscriptions is finished the knowbot is ready to start the auction.

Method5: A subscription of a late participant is registered.

Method6: A participant is unsubscribed.

Method7: The auctioneer puts up an item for auction, and then invites the participant to send their bids, starting from the base price MinB.

Method8: The auctioneer updates the current bid with the value of a valid new bid.

Method9: The auctioneer received a new bid higher than the bid he was waiting for, then it declares the item as sold.

Method10: If the waiting time for receiving a new bid expired the auctioneer declares the item sold.

The main Data Structure for the Participant is:
- item(Name,Me,MaxPrice): contains the information about the item;
- new_bid(Name,Me,NewB): contains the new offer of the participant.

class participant = {
in item_sold(Name,Winner,CurrB), item(Name,Me,MaxPrice);
out terminate
#
in subscribe(Part, Name, CreditNum);
send [auctioneer(_), subscribe(Part, Name, CreditNum)]
#
in unsubscribe(Part, Name);
send [auctioneer(_), unsubscribe(Part, Name)]
#
read item(Name, Me, MaxPrice);
in curr_bid(Name, Me, CurrB)
#
read item(Name, Me, MaxPrice);
in curr_bid(Name, Other, CurrB);
with {NewB is CurrB + 5, NewB < MaxPrice};
send [auctioneer(_), new_bid(Name, Me, NewB)]
#
in curr_bid(Name, Other, CurrB), item(Name, Me, MaxPrice);
with {CurrB >= MaxPrice};
out terminate
).

Class Participant

Method1: If the item has been sold the participant terminates.
Method2: The participant asks the auctioneer to subscribe the auction.
Method3: The participant asks the auctioneer to unsubscribe the auction.
Method4: The participant sends a new bid only if it is not the author of the current bid.
Method5: The participant sends a new bid if it can.
Method6: The participant terminates if it cannot propose a new bid higher than the current bid.

When a user wants a service of type auctioneer, he sends to the system the message "create(auctioneer(username))". This will cause the creation of an object with name auctioneer(username) (knowbot). When the auctioneer wants to start a bidding, he will send the message "time(Interval)" to his software representant, then he will wait for the signal of the auctioneer to start. The signal is the name of the item to sell ("up_item(Name)"). When the auctioneer sends "up_item(Name)", his knowbot inits his timer and starts to wait the subscriptions. This will make the timer to countdown until the "Interval" time. Each time a participant sends its subscription the auctioneer registers the name of the participant and its credentials. After the time interval set for the subscriptions, the auctioneer puts the item on sale sending new_item(Name, MinB, MaxBid, Timeout). From now on, each time a participant sends a new bid the auctioneer evaluates it, broadcasting the related communications to the participants.

When a user wants a service of type Participant he sends his credentials to the auctioneer, and the message "create(participant(username))" to the system; this will cause the creation of an object with name "participant(username)" (namely a knowbot). When the user sends the message item(Name, Me, MaxPrice) to its knowbot, the software representant will start to elaborate new bids, if this is allowed from the limit on the value of the top price.

Session

Here is an example of a session:

create(auctioneer(stefy), time(200)): sent by the auctioneer naming "stefy" to the system to create her knowbot;
send(auctioneer(stefy), up_item(picture)): sent by the auctioneer "stefy" to her knowbot for putting up the item "picture";
send(auctioneer(stefy), subscribe(davide, picture, 112298)): sent by the user “davide” to the auctioneer “stefy” for subscribing the auction;
create(participant(davide)): sent by the participant “davide” to ask for the creation of his knowbot;
send(auctioneer(stefy), subscribe(paolo, picture, 235161)): sent by the participant “paolo” to the auctioneer “stefy”, for subscribing the auction;
send(auctioneer(stefy), new_item(picture, 10, 40, 100)): sent by the auctioneer to his knowbot;
create(participant(paolo)): sent by the participant “paolo” to ask for his knowbot;
send(participant(paolo), item(picture, paolo, 53)): sent by the participant “paolo” to his knowbot to start the bidding;
send(participant(davide), item(picture, davide, 50)): sent by the participant “davide” to his knowbot for starting the bidding.

5.2.2 Version with multiple policies
We provide a participant with the choice of one among several policies for the generation of new bids, replacing the class participant in the previous program with the following three classes:

```plaintext
class participant = {
    in item_sold(Name, Me, CurrB), item(Name, Me, MaxPrice);
    send [_, victory(Name, Me, CurrB)];
    out terminate
}
```

```
class participant:incr = {
    read item(I, Me, MaxPrice);
    in curr_bid(I, Me, CurrB) #
    in curr_bid(Name, Other, CurrB), item(Name, Me, MaxPrice);
    send [_, lost(Name, Me, CurrB)];
    out terminate
    #
    read item(Name, Me, MaxPrice);
    in curr_bid(Name, Me, CurrB) #
    in curr_bid(Name, Other, CurrB), item(Name, Me, MaxPrice);
    with {CurrB > MaxPrice};
    out terminate
}.
```

```
class participant:double = {
    read item(I, Me, MaxPrice);
    in curr_bid(I, Other, CurrB);
    with {2 * CurrB < MaxPrice, NewB is 2 * CurrB};
    send [auctioneer(_), new_bid(I, Me, NewB)]
}.
```

The fourth method of the class participant of the first version has been removed and two new classes participant:incr and participant:double have been added. Each one of
these two classes comprises only one method, which defines two different policies. A user who wants a service of type participant can choose one of the two policies requiring a knowbot of type participant:incr, if he wants that a new bid is obtained incrementing of one unit his last bid, or a knowbot of type participant:double if he wants that a new bid is obtained doubling his last bid.

5.2.3 Interactive Version

In the following version both the auctioneer and the participants are provided with the interactive modality. Time functionalities are an external service provided from outside, and not directly to the auctioneer.

```plaintext
class auctioneer = {
  in time(Time);
  out waitsubscription(Time)
#
  in waitsubscription(Time), up_item(Name);
  send [participant(_), timeForSubscription(Name, Time)],
    [timer, alarm, Time, auctioneer(_), Name];
  out wait_cred(Name, Time), credL([])
#
  read wait_cred(Name, Time);
  in subscribe(Part, Name, CreditNum), credL(L);
  send [_], subscribed(Part, Name)];
  out credL([part(Part, CreditNum) | L]), accPart(Part)
#
  in ring(Name), wait_cred(Name, Time);
  send [participant(_), the_bidding_is_going_to_start];
  out maxTime(Time)
#
  in maxTime(Time), newTime(NTime);
  out maxTime(NTime)
#
  in subscribe(Part, Name, CreditNum), credL(L);
  send [_], ok(participant(Part), Name)];
  out credL([part(Part, CreditNum) | L]), accPart(Part)
#
  in unsubscribe(Part, Name), accPart(Part);
  send [_], ko(participant(Part), Name)]
#
  read maxTime(Time);
  in new_item(Name, MinB);
  send [_], curr_bid(Name, nobody, MinB), [timer, countdown(Time)],
    [timer, delete, Name], [timer, alarm, Time, auctioneer(_), Name];
  out item(Name), curr_bid(Name, nobody, MinB)
#
  read item(Name), accPart(WhoNew), maxTime(Time);
  in new_bid(Name, WhoNew, NewB), curr_bid(Name, Who, CurrB);
  with {NewB > CurrB};
  send [_], curr_bid(Name, WhoNew, NewB), [timer, countdown(Time)],
    [timer, delete, Name], [timer, alarm, Time, auctioneer(_), Name];
  out curr_bid(Name, WhoNew, NewB)
#
  in item(Name), curr_bid(Name, Who, CurrB), ring(Name);
  send [_], item_sold(Name, Who, CurrB)];
```
5.2.4 Interfaces for the Auction System

We have realized some tk interfaces for the auction users:

- **Auctioneer.etk** (Fig.4): this is the interface for an auctioneer. **Auctioneer.etk** allows the auctioneer to connect to ShaPE, to put an item on sale, to send messages to the participants and the observers.

- **Participant.etk** (Fig.5): this is the interface for a participant. **Participant.etk** allows a participant to connect to ShaPE, and to send messages to the auctioneer.

- **Observer.etk**: this is the interface for an Observer. **Observer.etk** allows an observer to connect to ShaPE to only observe the bidding.

5.3 Stock Exchange

Negotiations in a Stock Exchange [27] offer an ideal scenario for describing interactions among groups of users using ShaDe on a network.

Users in such a scenario can be divided in:

- **Clients**
- **Managers**
- **Brokers**

Let us suppose that \( n \) stock exchanges are active in the world. Each one of them support negotiations for securities.

A client is a person who wants to buy or sell securities, so he presents his order to a manager. A manager is a person who is in contact with brokers, and when he receives a request from a client he starts the negotiation for the client. For each received order, the Manager sends a request to a Broker for starting a negotiation.

The broker asks the stock exchanges for the negotiation, and waits for the answers. Once the offers have been collected and ordered, the broker will start a negotiation with the offers. If the order of the client can be satisfied, the broker notifies the trading of the requested securities both to the client and the counterpart of the negotiation. If the order of the client cannot be satisfied, the broker will wait until there will be the necessary conditions for a positive outcome of the negotiation.

Moreover, we suppose that the negotiation for the securities in a given stock exchange \( s-e \) will take in consideration the situation of the other \( n - 1 \) stock exchanges, that is, when a transaction for a supply or a demand is started in \( s-e \), the research of the securities will comprise also the other \( n - 1 \) stock exchanges, if needed. In each stock exchange both “local” and “external” (that is belonging to different stock exchanges) brokers will be active in parallel for the supply and demand of securities.

5.4 Stock Exchange in ShaDe

Modelling the Stock Exchange system in ShaDe we consider the following four classes:

- **Manager**
- **Broker**
- **Stock-exchange**
- **Timer**

Once some offers have been collected and ordered in a list, the broker agent will visit the list, and for each element he will start a negotiation. If an order of a client can be satisfied, the broker notifies the trading to both the client and his counterpart. If that order cannot be satisfied, the broker agent waits.

If the number of securities to sell or buy is less or equal than the first in the list, the broker sends a communication to his client and one to the counterpart. These communications notify the trading of the requested securities according to the quantity and price indicated in the considered
Figure 4. The auctioneer’s interface
Figure 5. The participant's interface
element of the list. If the order of the client have not been completely satisfied, the broker proceeds considering the second element of the list, and so on.

If no offer received by the broker satisfies the order, the remaining unsatisfied part of the order is inserted in the stock exchange chosen by the client, so that it will be satisfied later, if possible.

If the quantity of securities, that the broker has to buy or sell, is less than the sum of all the securities collected in the list, the broker, once the request of the client has been satisfied, sends the remaining securities to the manager. The manager will create a Broker for each request. This is done because when the broker asks the stock exchanges for securities, according to the conditions given by the client, every stock exchange which owns such a type of securities sends them to the Broker, and eliminates them from its database. If the request about the securities left unsold or not bought were simply reinserted in the stock exchange of origin, the requests could also result unsatisfied even if the conditions would be favorable. It could be possible, in fact, to have no negotiation in a stock exchange with requests for demand and requests for supply of the same type of securities.

The stock exchange, who receives a request for a supply or a demand, checks in its database the securities which could satisfy the request, and if this is the case sends them to the broker which sent the request, eliminating the security from its database.

The Data Structures for the Manager are:

- **security_demand(Sec,MaxPrice,Qa,Client,s_e(S))** data structure containing the information about the securities to buy, Sec is the type of the security, MaxPrice is the top price, Qa is the number of securities to buy, Client is the name of the client, s_e(S) is the stock exchange indicated by the client.

- **looking_for_buy(Sec,MaxPrice,Qa,Client)** message sent to the Stock-exchanges for a request of demand of securities, Sec is the type of the security, MaxPrice is the top price, Qa is the number of securities to buy, Client is the name of the client.

- **security_supply(Sec,MaxPrice,Qa,Client,s_e(S))** data structure containing the information about the securities to sell, Sec is the type of the security, MaxPrice is the top price, Qa is the number of securities to sell, Client is the name of the client, s_e(S) is the stock exchange indicated by the client.

- **looking_for_sell(Sec,MinPrice,Qa,Client)** message sent by the stock exchanges to sell the securities; Sec is the type of the security, MinPrice is the bottom price, Qa is the number of securities to sell, Client is the name of the client.

The main Data Structures for the Broker are:

- **next(Sec,s_e(S),b(Price,Qa,J,s_e(S1)),b(Pr1,Qa1,J1,s_e(S2)))** list of the requests of supply of securities sent by the stock exchanges, Sec is the type of the security, s_e(S) is the name of the Stock Exchange where the order will be stored if it is unsatisfiable at the moment, Price and Pr1 are the prices of the securities supplied, Qa and Qa1 are the quantities, J and J1 are the names of the related clients, s_e(S1) and s_e(S2) are the names of the Stock Exchanges of origin.

- **next(Sec,s_e(S),s(Price,Qa,J,s_e(S1)),s(Pr1,Qa1,J1,s_e(S2)))** list of the requests of demand sent by the stock exchanges, Sec is the type of the security, s_e(S) is the name of the Stock Exchange where the order will be stored if it is unsatisfiable at the moment, Price and Pr1 are the prices of the securities demanded, Qa and Qa1 are the quantities, J and J1 are the names of the related clients, s_e(S1) and s_e(S2) are the names of the Stock Exchanges of origin.

- **buy(Sec,Price,Qa,Client)** message sent to ask a Stock Exchange to insert in its own database the request for the supply of the securities Sec.

- **sell(Sec,Price,Qa,Client)** message to ask a Stock Exchange to insert in its own database the request for the demand of the securities Sec.

The Data Structures for the Stock-exchange are:

- **sell(Sec,Price,Qa,J)** is the request for the supply of Qa securities of type Sec in a Stock Exchange. Sec is the type of the security, Price is the price of the security, Qa is the
available quantity of securities, \( J \) is the name of the client.
- \( \text{buy}(\text{Sec}, \text{Price}, \text{Qa}, J) \) is the request for the demand of \( \text{Qa} \) securities of type \( \text{Sec} \) in a Stock Exchange. \( \text{Sec} \) is the type of the security, \( \text{Price} \) is the price of the security, \( \text{Qa} \) is the available quantity of securities, \( J \) is the name of the client.
- \( \text{stock\_to\_sell}(\text{Client}, \text{Sec}, \text{Price}, \text{Qa}, J) \) is the message sent by a Stock Exchange to the broker, who represents \( \text{Client} \), to ask for \( \text{Qa} \) securities of type \( \text{Sec} \) with price \( \text{Price} \), and belonging to \( J \).
- \( \text{stock\_to\_buy}(\text{Client}, \text{Sec}, \text{Price}, \text{Qa}, J) \) is the message sent by a Stock Exchange to the broker, who represents \( \text{Client} \), to offer \( \text{Qa} \) securities of type \( \text{Sec} \) with price \( \text{Price} \) and belonging to \( J \).

**Program**

```plaintext
class manager = {
  in security_demand(Sec, MaxPr, Qa, Client, s_e(S));
  send [s_e, looking_for_buy(Sec, MaxPr, Qa, Client)];
  create (broker(Client,Sec), wait_buy(Sec, MaxPr, Qa),
  next(Sec,s_e(S),b(0,0,0,0),b(999,9,9,9)),
  (timer(Client,Sec),begin(Client,Sec))
  #
  in security_supply(Sec, MinPr, Qa, Client, s_e(S));
  send [s_e, looking_for_sell(Sec, MinPr, Qa, Client)];
  create (broker(Client,Sec), wait_sell(Sec, MinPr, Qa),
  next(Sec,s_e(S),s(999,9,9,9),s(0,0,0,0)),
  (timer(Client,Sec),begin(Client,Sec))
};
```

**Class Manager**

**Method 1:** When the manager receives a request for a security demand, he creates a broker who will manage the negotiation for the security, and sends a message to the Stock Exchange to start the negotiation.

**Method 2:** When the manager receives a request for a security supply, he creates a broker who will manage the negotiation for the security, and sends a message to the Stock Exchange to start the negotiation.

```plaintext
class broker = {
  read wait_buy(Sec, MaxPr, Qa);
  in next(Sec,s_e(S),b(Pr3,Qa3,J3,s_e(S3)),b(Pr4,Qa4,J4,s_e(S4))),
  stock_to_buy(Sec,Pr2,Qa2,J2,s_e(S2));
  with { Pr2<Pr4, Pr2>=Pr3 };
  out next(Sec,s_e(S),b(Pr3,Qa3,J3,s_e(S3)),b(Pr2,Qa2,J2,s_e(S2))),
  stock_to_buy(Sec,Pr2,Qa2,J2,s_e(S2));
  #
  in wait_buy(Sec,MaxPr,Qa),end(Client,Sec);
  send [s_e, end_buy(Client,Sec)];
  out order_buy(Sec,MaxPr,Qa,Client)
  #
};
```

UBLCS-96-5 20
in next(Sec,s_e(S),b(0,0,0,0),b(999,9,9,9)),
    order_buy(Sec,MaxPr,Qa,Client);

send s_e(S),buy(Sec,MaxPr,Qa,Client,s_e(S));

out terminate
#
in next(Sec,s_e(S),b(0,0,0,0),b(Pr1,QA1,J1,s_e(S1))),
    next(Sec,s_e(S),b(Pr1,QA1,J1,s_e(S1)),b(Pr2,QA2,J2,s_e(S2))),
    order_buy(Sec,Price,Qa,Client);

with { Qa>QA1, QA3 is QA-QA1 };

send (s_e(S1),buy_done(Sec,QA1,Pr1,J1)),
    (s_e(S),sell_done(Sec,QA1,Pr1,Client));

out next(Sec,s_e(S),b(0,0,0,0),b(Pr2,QA2,J2,s_e(S2))),
    order_buy(Sec,Price,QA3)
#
in next(Sec,s_e(S),b(0,0,0,0),b(Pr1,QA1,J1,s_e(S1))),
    next(Sec,s_e(S),b(Pr1,QA1,J1,s_e(S1)),b(Pr2,QA2,J2,s_e(S2))),
    order_buy(Sec,Price,Qa,Client);

with { QA=:=QA1 };

send (s_e(S1),buy_done(Sec,QA1,Pr1,J1)),
    (s_e(S),sell_done(Sec,QA1,Pr1,Client));

out sell_revoke(Sec),next(Sec,s_e(S),
    b(0,0,0,0),b(Pr2,QA2,J2,s_e(S2)))
#
in next(Sec,s_e(S),b(0,0,0,0),b(Pr1,QA1,J1,s_e(S1))),
    next(Sec,s_e(S),b(Pr1,QA1,J1,s_e(S1)),b(Pr2,QA2,J2,s_e(S2))),
    order_buy(Sec,Price,Qa,Client);

with { QA<QA1, QA3 is QA1-QA };

send (s_e(S1),buy_done(Sec,QA,Pr1,J1)),
    (s_e(S),sell_done(Sec,QA,Pr2,Client));

out sell_revoke(Sec),
    next(Sec,s_e(S),b(0,0,0,0),b(Pr2,QA3,J1,s_e(S1))),
    next(Sec,s_e(S),b(Pr2,QA3,J1,s_e(S1)),b(Pr2,QA2,J2,s_e(S2)))
#
read sell_revoke(Sec);

in next(Sec,s_e(S),b(0,0,0,0),b(Pr2,QA2,J2,s_e(S2))),
    next(Sec,s_e(S),b(Pr2,QA2,J2,s_e(S2)),b(Pr3,QA3,J3,s_e(S3)));

send [manager,security_supply(Sec,Pr2,QA2,J2,s_e(S2))];

out next(Sec,s_e(S),b(0,0,0,0),b(Pr3,QA3,J3,s_e(S3)))
#
in sell_revoke(Sec),next(Sec,s_e(S),b(0,0,0,0),b(999,9,9,9));
out terminate
#
read wait_sell(Sec,MinPrice,Qa);

in next(Sec,s_e(S),s(Pr3,Qa3,J3,s_e(S3)),s(Pr4,Qa4,J4,s_e(S4))),
stock_to_sell(Sec,Pr2,Qa2,J2,s_e(S2));

with { Pr2=<Pr3, Pr2>Pr4 }; 

out next(Sec,s_e(S),s(Pr3,Qa3,J3,s_e(S3)),s(Pr2,Qa2,J2,s_e(S2))),
next(Sec,s_e(S),s(Pr2,Qa2,J2,s_e(S2)),s(Pr4,Qa4,J4,s_e(S4)))
#
in wait_sell(Sec,MinPrice,Qa),end(Client,Sec);

send [s_e,end_sell(Client,Sec)];

out order_sell(Sec,MinPrice,Qa,Client)
#
in next(Sec,s_e(S),s(999,9,9,9),s(0,0,0,0)),
order_sell(Sec,MinPrice,Qa,Client);

send [s_e(S),sell(Sec,MinPrice,Qa,Client,s_e(S))];

out terminate
#
in next(Sec,s_e(S),s(999,9,9,9),s(Pr2,Qa2,J2,s_e(S2))),
next(Sec,s_e(S),s(Pr2,Qa2,J2,s_e(S2)),s(Pr3,Qa3,J3,s_e(S3))),
order_sell(Sec,MinPrice,Qa,Client);

with { Qa>Qa2, Qa1 is Qa-Qa2 };

send [_,sell_done(Sec,Qa2,Pr2,J2,s_e(S2))],
[_,buy_done(Sec,Qa2,Pr2,Client,s_e(S))];

out order_sell(Sec,MinPrice,Qa1),
next(Sec,s_e(S),s(999,9,9,9),s(Pr3,Qa3,J3,s_e(S3)))
#
in next(Sec,s_e(S),s(999,9,9,9),s(Pr2,Qa2,J2,s_e(S2))),
next(Sec,s_e(S),s(Pr2,Qa2,J2,s_e(S2)),s(Pr3,Qa3,J3,s_e(S3))),
order_sell(Sec,MinPrice,Qa,Client);

with { Qa:=:=Qa2 };

send [_,sell_done(Sec,Qa,Pr2,J2,s_e(S2))],
[_,buy_done(Sec,Qa,Pr2,Client,s_e(S))];

out order_sell(Sec,MinPrice,Qa1),
next(Sec,s_e(S),s(999,9,9,9),s(Pr3,Qa3,J3,s_e(S3)))
#
in next(Sec,s_e(S),s(999,9,9,9),s(Pr2,Qa2,J2,s_e(S2))),
next(Sec,s_e(S),s(Pr2,Qa2,J2,s_e(S2)),s(Pr3,Qa3,J3,s_e(S3))),
order_sell(Sec,MinPrice,Qa,Client);
with { Qa<Qa2, Qa1 is Qa2-Qa };

send [_, sell_done(Sec, Qa, Pr2, J2, s_e(S2))],
    [_, buy_done(Sec, Qa, Pr2, Client, s_e(S))];

out next(Sec, s_e(S), s(999, 9, 9, 9), s(Pr2, Qa1, J2, s_e(S2))),
    next(Sec, s_e(S), s(Pr2, Qa1, J2, s_e(S2)), s(Pr3, Qa3, J3, s_e(S3))),
    buy_revoke(Sec)
# read buy_revoke(Sec);

in next(Sec, s_e(S), s(999, 9, 9, 9), s(Pr2, Qa2, J2, s_e(S2))),
    next(Sec, s_e(S), s(Pr2, Qa2, J2, s_e(S2)), s(Pr3, Qa3, J3, s_e(S3)));

send [manager, security_demand(Sec, Pr2, Qa2, J2, s_e(S2))];

out next(Sec, s_e(S), s(999, 9, 9, 9), s(Pr3, Qa3, J3, s_e(S3)))
# in buy_revoke(Sec), next(Sec, s_e(S), s(999, 9, 9, 9), s(0, 0, 0, 0));

out terminate
}.

Class Broker

Method1: The broker waits for the supply offers ("stock_to_buy(...)") coming from the Stock Exchanges, and it collects the offers in the list named "next".

Method2: The broker receives the signal ("end(...)") from the timer. It sends a message to the Stock Exchanges ("end_buy(...)"), to indicate that he cannot receive offers anymore, and then he starts the negotiation.

Method3: If the broker has received no supply offer, the securities have to be reinserted in the Stock Exchange indicated by the client sending the message ("buy(...)") to the Stock Exchange. The broker has then terminated his work.

Method4: The broker has bought Qa1 securities, and tries to buy the remaining Qa3, so it informs the client and its counterpart of the partial negotiation ("buy_done(...)" and "sell_done(...)") and it updates the list of the offers.

Method5: The broker has bought all the required securities, and the last group of securities was the exact amount needed, so he informs the client ("buy_done(...)"), and his counterpart ("sell_done(...)") that the negotiation is finished, and starts to revoke the offers that he had collected.

Method6: The broker has bought all the required securities, but the last group of securities was too large, so he informs the client ("buy_done(...)") and his counterpart ("sell_done(...)") that the negotiation is finished, and starts to revoke the remaining part of the last group of securities, and all the other collected offers.

Method7: The supply offer indicated in the first element of the list "next(...)" is revoked and sent to the manager as a new order ("security_supply(...)”). This will cause the creation of a new broker who will manage the negotiation for the order. This new broker will check if somebody has inserted demand offers in the Stock Exchange, in such a case he will start the negotiation, otherwise he will insert the offer in the Stock Exchange s_e(S).

Method8: The broker has revoked all the offers then he terminates.

Method9: The broker waits for the demand offers coming from the Stock Exchanges ("stock_to_sell(...)”). It collects the offers in the list "next".

Method10: The broker receives the stop signal from the timer, so it sends a message to the stock exchanges to indicate that he doesn’t want to receive any other offer, then it starts the negotiation.
Method 11: If the broker has received no demand offers, the securities have to be inserted in the stock exchange indicated by the client, sending a message to the given stock exchange ("sell(...)”). The broker has then terminated its work.

Method 12: The broker sells $Q_{a2}$ securities and tries to sell the last $Q_{a1}$, so it informs the client and its counterpart of the partial negotiation ("sell_done(...)”, “buy_done(...)”), and updates the list of the offers.

Method 13: The broker has sold all the required securities, and the last group of securities was the exact amount needed, so he informs the client and his counterpart that the negotiation is finished, and starts to revoke the offers that he has collected.

Method 14: The broker has sold all the required securities, but the last group of securities was too large, so he informs the client and his counterpart (“sell_done(...)”, “buy_done(...)”) that the negotiation is finished, and starts to revoke the remaining part of the last group of securities, and all the other collected offers.

Method 15: The demand offer indicated in the first element of the list “next(...)” is revoked and sent to the manager as a new order (“security_demand(...)”). This will cause the creation of a new broker who will manage the negotiation for the order. This new broker will check if somebody has inserted supply offers in the Stock Exchange, in such a case he will start the negotiation, otherwise he will insert the offer in the Stock Exchange $s_e(s)$.

Method 16: The broker has revoked all the offers then he terminates.

class s_e = {
    read looking_for_buy(Sec,MaxPrice,Qa,Client);

    in sell(Sec,Pr2,Qa2,J2,s_e(S))
    
    with { MaxPrice>=Pr2 };

    send [broker(Client,Sec),stock_to_buy(Sec,Pr2,Qa2,J2,s_e(S))]

    # in looking_for_buy(Sec,MaxPrice,Qa,Client),end_buy(Client,Sec);

    # read looking_for_sell(Sec,MinPrice,Qa,Client);

    in buy(Sec,Pr,Qa,J,s_e(S))

    with { MinPrice=<Pr };

    send [broker(Client,Sec),stock_to_sell(Sec,Pr,Qa,J1,s_e(S))]

    # in looking_for_sell(Sec,Pr,Qa,Client),end_sell(Client,Sec)
};

Class Stock-Exchange

Method 1: When the stock exchange receives a request from the broker of Client ("looking_for_buy(...)”), it sends the demand offers to Client, if any.

Method 2: When the stock exchange receives the message “end_buy(...)” no other demand offer has to be sent to Client.

Method 3: When the stock exchange receives a supply request ("looking_for_sell(...)”) from Client, it sends the supply offers to Client, if any.
Method 4: When the stock exchange receives the message “end_sell(..)” no other supply offer has to be sent to Client.

```plaintext
class Timer = {
    in begin(Client,Sec);
    out wait(20000,Client,Sec)
    # in wait(X,Client,Sec);
    with { X>0, Y is X-1 }; 
    out wait(Y,Client,Sec) 
    # in wait(0,Client,Sec);
    send [broker(Client,Sec),end(Client,Sec)];
    out terminate 
}.
```

Class Timer

Method 1: When the timer is created it starts to wait for the given time interval.
Method 2: It waits until the end of the time interval.
Method 3: When the time interval expires the timer sends the communication “end(..)” to Client.

6 Semantics by translation

6.1 LO
Linear Object (LO) [6, 7] is a coordination language based on Linear Logic. Its coordination entities are multisets; the communication media is called the forum, and supports essentially broadcasting of a message to all active multisets.

A computation in LO can be thought of as the evolution of a system of communicating agents; an agent state is represented by a multiset of atoms contained in a private context; agent’s state transitions are expressed by multiset rewritings, and can be either state transformations, or creations of new agents, or terminations.

The syntax of the language LO uses three connectives of Linear Logic: @ (“par”), & (“with”), and T (“top”), and the Linear Implication “<>-”. The classes of Linear formulae Goal and Method, recursively built from the class A of atoms, can be described with the following BNF grammar:

```
Goal = A | Goal @ Goal | Goal & Goal | T
Method = Head @ [Tell] | @ [P_goal] | <><- Goal.
Head = A | Head @ Head
Tell = "A | "A @ Tell
```

Some atoms in the left side of a method M can be marked by the tell marker “"," which denotes a form of broadcasting communication. P_goal is a conjunction of Prolog goals.

An LO program is a set of methods and an LO query is a pair written \((P, G)\) where \(P\) is a program and \(G\) is a ground goal.

6.2 Mapping ShaDe onto LO
In this section we show how a ShaDe program can be mapped onto an LO program.
We saw in Sec.3.1 that a ShaDe program is a set of classes, and each class is a set of methods. Let $P'$ be a ShaDe program. $P'$ can be transformed into an LO program $P$ passing through the following sequence of steps.

- **Step1**: elimination of the class structure.
  1. for each class $C$ which is subclass of the class $C'$ we add the methods of $C'$ to the methods of the class $C$.
  2. we substitute each method $M$ belonging to the class $C_{name}$

\[
M =
\begin{align*}
\text{read tokens;}
\text{in tokens';}
\text{with \{ seq_program \};}
\text{send messages;}
\text{create objects;}
\text{out actions}
\end{align*}
\]

with the method $M'$ obtained from $M$ replacing

\[
\text{read tokens;}
\]

with

\[
\text{read tokens,C_{name};}
\]

In this way the information regarding the class is inserted into the syntactic structure of methods.

Eliminating then the class structure we obtain a program configured as a sequence of methods.

- **Step2**: elimination of the read field in a method.
  1. We substitute each method $M$ of the form:

\[
\begin{align*}
\text{read tokens;}
\text{in tokens';}
\text{with \{ seq_program \};}
\text{send messages;}
\text{create objects;}
\text{out tokens''}
\end{align*}
\]

with the method $M'$ obtained from $M$ cancelling the read field and replacing the fields

\[
\begin{align*}
\text{in tokens';}
\text{out tokens''}
\end{align*}
\]

with

\[
\begin{align*}
\text{in tokens',tokens;}
\text{out tokens'',tokens;}
\end{align*}
\]

2. We substitute each method $M$ of the form:

\[
\begin{align*}
\text{read tokens;}
\text{in tokens';}
\text{with \{ seq_program \};}
\text{send messages;}
\text{create objects;}
\text{out terminate}
\end{align*}
\]

with the method $M'$ obtained from $M$ cancelling the read field and replacing the field

\[
\text{in tokens'};
\]

with

\[
\text{in tokens',tokens;}
\]

3. We substitute each method $M$ of the form:

\[
\text{read tokens;}
\]
with the method $M'$ obtained from $M$ cancelling the read field and replacing the field

\[ \text{in tokens';} \]

with

\[ \text{in tokens', tokens;} \]

and adding the field

\[ \text{out tokens'.} \]

- **Step3**: termination
  we substitute the keyword termination with the symbol $\top$.

- **Step4**: Communication translation
  A message in the send field can have one of the following forms:
  a) \[[\_ \text{tokens}]\]
  b) \[[\text{objectType}(\_), \text{tokens}]\]
  c) \[[\text{ObjectName}, \text{token}]\]
  In the a) case we replace \[[\_ \text{tokens}]\] with $\sim \text{token}$, for each token in tokens.
  In the b) case we replace \[[\text{objectType}(\_), \text{tokens}]\] with $\sim \text{send(objectType,token)}$, for each token in tokens.
  In the c) case we replace \[[\text{ObjectName}, \text{tokens}]\] with $\sim \text{send(objectName,token)}$, for each token in tokens.

- **Step5**: Creation translation
  A request for a creation in the create field has the following form:
  \[[\text{ObjectType,ObjectName}, \text{tokens}]\]
  We replace \[[\text{ObjectType,ObjectName}, \text{tokens}]\] with $\sim \text{req(objectType,ObjectName,tokens)}$.

- **Step6**: Method structure transformation
  We denote with $\text{conj}$ the following constructor of disjunctions:
  let $A$ be a multiset of tokens, that is $A$ has the form $A = \{ a_1, \ldots, a_n \}$ then
  \[
  \text{conj}(A) = a_1 \oplus \cdots \oplus a_n \quad \text{if } n > 1
  \\
  \text{conj}(a_1) = a_1 \quad \text{if } n = 1
  \]
  Let us suppose that the steps described above have been applied to the methods of a program. We substitute all the resulting methods, $M$, of the form:
  \[
  \text{in tokens';} \]
  \[
  \text{with \{} \text{seq_program} \};
  \]
  \[
  \text{send messages;}
  \]
  \[
  \text{create objects;}
  \]
  \[
  \text{out tokens';}
  \]
  with the following method:
  \[
  \text{conj(tokens')} \oplus \{ \text{seq_program} \} \oplus \text{conj(messages)} \oplus \text{conj(objects)} \oplus \text{conj(tokens')}.
  \]

- **Step7**: Interpretation of the communications
  The following methods are added for each class at compilation time:
  \[
  \text{send(prototype\_name,token)} \oplus \text{prototype\_name} \oplus \text{token}.
  \]
  \[
  \text{send(destIdentifier,token)} \oplus \text{myIdentifier} \oplus \{ \text{match(destIdentifier, myIdentifier)} \} \oplus \text{myIdentifier} \oplus \text{token}.
  \]

- **Step8**: Interpretation of the creations
  For each creation in the methods of the ShaDe program is added a method of the following form:
Acknowledgments. We thank F. Panzieri for the suggestion of modeling the Auction System. Partial support for this work was provided by the Commission of European Communities under ESPRIT Programme Basic Research Project 9102 (COORDINATION) and by Italian CNR - Comitato Scienza e Tecnologia dell’Informazione, under contract 94.00417.CT12.

References


